2001-10-01

Efficient Algorithms for Elliptic Curve Cryptosystems on Embedded Systems

Adam D. Woodbury
Worcester Polytechnic Institute

Efficient Algorithms for Elliptic Curve Cryptosystems on Embedded Systems

by
Adam D. Woodbury

A Thesis
submitted to the Faculty
of the
Worcester Polytechnic Institute
In partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical Engineering
by

September, 2001

Approved:

Dr. Christof Paar Thesis Advisor ECE Department	Dr. Berk Sunar Thesis Committee ECE Department
Dr. William Martin Thesis Committee Mathematical Sciences Department	Dr. John Orr Department Head ECE Department

Abstract

This thesis describes how an elliptic curve cryptosystem can be implemented on low cost microprocessors without coprocessors with reasonable performance. We focus in this paper on the Intel 8051 family of microcontrollers popular in smart cards and other cost-sensitive devices, and on the Motorola Dragonball, found in the Palm Computing Platform. The implementation is based on the use of the Optimal Extension Fields $GF((2^8 - 17)^{17})$ for low end 8-bit processors, and $GF((2^{13} - 1)^{13})$ for 16-bit processors.

Two advantages of our method are that subfield modular reduction can be performed infrequently, and that an adaption of Itoh and Tsujii's inversion algorithm may be used for the group operation. We show that an elliptic curve scalar multiplication with a fixed point, which is the core operation for a signature generation, can be performed in a group of order approximately 2^{134} in less than 2 seconds on an 8-bit smart card. On a 16-bit microcontroller, signature generation in a group of order approximately 2^{169} can be performed in under 700 milliseconds. Unlike other implementations, we do not make use of curve parameters defined over a subfield such as Koblitz curves.

Preface

This work details the research I conducted at Worcester Polytechnic Institute in pursuit of my Master's degree.

I would first like to thank Prof. Christof Paar, who has been my advisor, mentor, and friend since I began my studies with him. Working with Prof. Paar has taken me to foreign lands, presented amazing opportunities, and introduced me to the legends of the field. It was solely my desire to do further work in cryptography with Prof. Paar that led me to continue my studies at WPI.

I would like to thank my Thesis committee, Prof. Berk Sunar and Prof. William Martin for their time and suggestions. I am grateful for their acceptance of my unreasonable requests and timeline.

I would like to thank Dan Bailey, Brendon Chetwynd, Adam Elbirt, Jorge Guajardo, Carleton Jillson, Andre Weimerskirch, and Thomas Wollinger for such a great atmosphere in and surrounding the Cryptography lab.

Finally, and most importantly I would like to dedicate this thesis to Sandra, my wife. She was willing to live with me in a graduate student's life (and meager pay) so that I was able to study for my Master's degree. She has sacrificed more than I care to mention here, and it is my intention to make it up to her some day. I thank her for this chance, as this graduate work has been more rewarding than I could have imagined.

Adam D. Woodbury

Contents

List of Tables

Chapter 1

Introduction

1.1 Why ECC?

The challenge addressed in this thesis is to implement a public-key digital signature algorithm on embedded systems which neither introduces performance problems nor requires additional hardware. To approach this problem, we turn to the computational savings made available by elliptic curve cryptosystems. An elliptic curve cryptosystem relies on the assumed hardness of the Elliptic Curve Discrete Logarithm Problem (ECDLP) for its security. An instance of the ECDLP is posed for an elliptic curve defined over a finite field $GF(p^m)$ for p a prime and m a positive integer. The rule to perform the elliptic curve group operation can be expressed in terms of arithmetic operations in the finite field; thus the speed of the field arithmetic determines the speed of the cryptosystem.

Two target platforms are chosen to represent the broad spectrum of embedded systems. The first target is an 8-bit microcontroller, the Intel 8051, derivatives of which are on many popular smart cards such as the Infineon SLE44C200 and Philips

82C852. The second target is a 16-bit microcontroller, the Motorola MC68328, found in the popular Palm and Visor handheld PDAs. The implementation is focused on efficient software algorithms for finite field arithmetic and efficient ECC point multiplication with a fixed point on embedded μPs.

In Chapter 5, we compare the finite field arithmetic performance offered on an 8-bit microcontroller by three different types of finite field which have been proposed for elliptic curve cryptosystems (ECCs): binary fields $GF(2^n)$, binary composite fields $GF((2^n)^m)$, and finally Optimal Extension Fields (OEFs): $GF(p^m)$ for p a pseudo-Mersenne prime, m chosen so that an irreducible binomial exists over $GF(p)$. Our results show that core field arithmetic operations in $GF(2^n)$ lag behind the other two at a ratio of 5:1. The arithmetic offered by OEFs and composite fields is comparable in performance. However, the recent result of Gaudry, Hess, and Smart [GHS00] has shown that the ECDLP can be easily solved when certain composite fields are used. Thus, in the main part of this thesis we present the results of applying OEFs to the construction of ECCs to calculate digital signatures on embedded platforms within a reasonable processing time with no need for additional coprocessor hardware.

1.2 Why embedded platforms?

> "While the Internet creates a new cyberspace separate from our physical world, technological advances will enable ubiquitous networked computing in our day-to-day lives. The power of this ubiquity will follow from the *embedding* of computation and communications in the physical world– that is, embedded devices with sensing and communication capabilities that enable distributed computation [EGH00]."

Technological improvements have made possible the development of power-

ful, low cost and low power microprocessors. These devices have already begun to be incorporated into our every day lives, each responsible for keeping track of our addresses, appointments, messages, etc. Furthermore, while these devices become increasingly connected, the communication channels between them remain highly insecure. Thus we have a situation that demands the use of cryptography, but does not possess quite enough power to run the traditional algorithms in a reasonable amount of time. Constrained environments present a difficult challenge, where every possible optimization must be used simply to enable the very use of security.

As mentioned above, some examples of these embedded systems are already prevalent, while others are still on the horizon. A prime example of an embedded system in need of security is the cell phone. The popularity of cell phones is growing, and FCC has required GPS capabilities be added by 2005 to facilitate location by 911 services. Without proper security, this functionality could be exploited to enable the accurate and automatic tracking of individuals by those with malicious intent. In other areas, the emergence of local wireless communication technologies such as Bluetooth, IRDA, and IEEE 802.11 is enabling interconnection among smaller devices. On the horizon, efforts are being made to interconnect household appliances and provide connectivity to devices within automobiles. Many of these systems need security to enable the protection of data privacy and integrity.

Chapter 2

Background

2.1 Elliptic Curve Cryptosystems

The use of elliptic curve cryptosystems is relatively new. They were introduced independently by Victor Miller [Mil86] and Neil Koblitz [Kob87], and have since been a popular research area. The reason elliptic curves (EC) are so tempting for cryptographic use is because the key lengths are significantly shorter than those of public-key (PK) systems based on the integer factorization or finite field discrete logarithm problem. According to the IEEE 1363 standards specification [IEE00], an RSA key of 1024 bits is considered security equivalent to an elliptic curve cryptosystem with keys of 172 bits. The cost of complex mathematical operations increases significantly with the length of their operands.

The primary operation in an ECC is scalar-point multiplication $C = kP$, where P is a point on the curve and k is an integer. The multiplication is performed using the group operation detailed in the following section.

2.1.1 Group Operation

An elliptic curve can be viewed as the set of all solutions to an equation of the form

$$y^2 = x^3 + ax + b. \tag{2.1}$$

It is possible to turn this set of points into an Abelian group; the operation is called "point addition." This operation adds two curve points, and results in another point on the curve. We restrict our attention to points all of whose coordinates lie in some given finite field $GF(q)$ containing a and b. These are referred to as $GF(q)$-rational points. We denote by $E(GF(q))$ the group of all $GF(q)$-rational points and use $\#E(GF(q))$ for the order of this group. Using an ECC for signatures involves the repeated application of the group law. The group law using affine coordinates is shown below [Men93].

Definition 2.1: If $P = (x_1, y_1) \in E(GF(q))$, then $-P = (x_1, -y_1)$. If $Q = (x_2, y_2) \in E(GF(q)), Q \neq -P$, then $P + Q = (x_3, y_3)$, where

$$x_3 = \lambda^2 - x_1 - x_2, \tag{2.2}$$

$$y_3 = \lambda(x_1 - x_3) - y_1, \tag{2.3}$$

$$\lambda = \begin{cases} \frac{y_2 - y_1}{x_2 - x_1}, & \text{if } P \neq Q, \\ \\ \frac{3x_1^2 + a}{2y_1}, & \text{if } P = Q. \end{cases} \tag{2.4}$$

For fields of characteristic two or three, the expressions are slightly different.

The λ term is calculated depending on the relationship of P and Q. If they are equal, then a point doubling is performed, using the second equation. Note that λ is undefined if the points are additive inverses, or if either point is zero. These conditions must be examined before the group operation is performed.

2.2 Finite Fields

To implement an ECC, an implementor must select an underlying finite field in which to perform arithmetic calculations. A finite field is identified with the notation $GF(p^m)$ for p a prime and m a positive integer. It is well known that there exists a finite field for all primes p and positive integers m. Any such field is isomorphic to $GF(p)[x]/(P(x))$, where $P(x) = x^m + \sum_{i=0}^{m-1} p_i\, x^i, p_i \in GF(p)$, is a monic irreducible polynomial of degree m over $GF(p)$. In the following, each residue class will be identified with the unique polynomial of least degree in this class.

Various finite fields admit the use of different algorithms for arithmetic. Unsurprisingly, the choices of p, m, and $P(x)$ can have a dramatic impact on the performance of the ECC. In particular, there are generic algorithms for arithmetic in an arbitrary finite field and there are specialized algorithms which provide better performance in finite fields of a particular form. In the following, we briefly describe field types proposed for ECC.

2.2.1 Binary Fields

Implementors designing custom hardware for an ECC often choose $p = 2$ and $P(x)$ to be a trinomial or pentanomial. Such choices of irreducible polynomial lead to efficient methods for extension field modular reduction. We will refer to this type of field as a "binary field," in accordance with [IEE00]. The elements of the subfield $GF(2)$ can be represented by the logical signals 0 and 1. In this way, it is possible to construct fast and area efficient hardware circuits to perform the finite field arithmetic. Binary fields are also popular for software implementations of ECC.

2.2.2 Binary Composite Fields

In software, the choice of parameters varies considerably with the wide array of available microprocessors. Many authors have suggested the use of $p = 2$ and m a composite number, e.g. [DBV+96, GP97]. In this case, the field $GF(2^m)$ is isomorphic to $GF((2^s)^r)$, for $m = sr$ and we call this a "composite field." Then multiplication and inversion in the subfield $GF(2^s)$ can be efficiently performed by index table look-up if s is not too large. In turn, these operations in the extension field $GF((2^s)^r)$ are computed using arithmetic in the subfield. As in the binary field case, the irreducible polynomials for both the subfield and the extension field are chosen to have minimal weight. This approach can provide superior performance when compared to the case of binary fields. *However, a recent attack against ECCs over composite fields [GHS00] makes their use in practice questionable.*

2.2.3 Prime Fields

Prime fields, where $m = 1$ are perhaps the most obvious finite fields to use. For ECC, a typical prime is chosen to be larger than 2^{160}, and must be stored in multiple computer words. The problem with this representation is that during computation, the carries between words must be propagated, and the reduction modulo p must be performed over several words. There has been a large amount of research dealing with methods for doing long-number multi-precision arithmetic efficiently. Perhaps the most popular method in this context is based on Montgomery reduction [Mon85].

2.2.4 Optimal Extension Fields

An alternative construction is to use optimal extension fields (OEFs) [BP98], defined as follows. Choose p of the form $2^n \pm c$, for n, c arbitrary positive integers, where $log_2(c) \leq \lfloor \frac{1}{2} n \rfloor$. In this case, one chooses p of appropriate size to use the multiply instructions available on the target microcontroller. In addition, m is chosen so that an irreducible binomial $P(x) = x^m - \omega$ exists, $\omega \in GF(p)$. The algorithmic and implementation details for OEFs will be discussed in Chapters 4, 5 and 6.

To generate "good" elliptic curves over OEFs there are two basic approaches. The first one is based on the use of a curve defined over $GF(p)$ using the method in [BSS99, Section VI.4]. The second, more general, method uses Schoof's algorithm together with its improvements.

Chapter 3

Previous Work

3.1 Previous Work

This section reviews some of the most relevant previous contributions. It has been long recognized that efficient finite field arithmetic is vital to achieve acceptable performance with ECCs. Before an attack was published making their use in practice questionable, many implementors chose even-characteristic finite fields with composite extension degree.

A paper due to De Win et al. [DBV+96] analyzes the use of fields $GF((2^n)^m)$, with a focus on $n = 16$, $m = 11$. This construction yields an extension field with 2^{176} elements. The subfield $GF(2^{16})$ has a Cayley table of sufficiently small size to fit in the memory of a workstation. Optimizations for multiplication and inversion in such composite fields of characteristic two are described in [GP97].

Schroeppel et al. [SOOS95] report an implementation of an elliptic curve analogue of Diffie-Hellman key exchange over $GF(2^{155})$. The arithmetic is based on a polynomial basis representation of the field elements. Another paper by De Win

et.al. [DMPW98] presents a detailed implementation of elliptic curve arithmetic on a desktop PC, using finite fields of the form $GF(p)$ and $GF(2^n)$, with a focus on its application to digital signature schemes. For ECCs over prime fields, it is possible to use projective coordinates to eliminate the need for inversion, along with a balanced ternary representation of the multiplier.

The work in [Bai98, BP98] introduces OEFs and provides performance statistics on high-end RISC workstations. Odd characteristic extension fields for use in cryptography were introduced independently by Preda Mihăilescu in a rump session in [Mih97]. A paper extending the work on OEFs appears in [KMKH99]. In this paper, sub-millisecond performance on high-end RISC workstations is reported. Further, the authors achieve an ECC performance of 1.95 msec on a 400 MHz Pentium II with a field of order 2^{186}. Reference [BP01] describes the Itoh-Tsujii inversion algorithm for OEFs which is used in this contribution.

In [NM96], Naccache and M'Raïhi provide an overview of smart cards with cryptographic capabilities, including a discussion of general implementation concerns on various types of smart cards. In [NMWdP95] a zero-knowledge system on an 8-bit microprocessor without a coprocessor is presented.

In a white paper [Cer98], Certicom Corp. provides an implementation overview for an ECC defined over $GF(2^{163})$ on smart card CPUs without cryptographic coprocessors.

Chapter 4

Relevant Algorithms

The key to a successful and efficient implementation of a cryptosystem is the choice of algorithms to optimize the arithmetic. While a multitude of algorithms exist, it is important to carefully choose the best combination. In this chapter, we will discuss the primary algorithms used in this implementation.

4.1 Karatsuba Multiplication

Extension field multiplication is the most costly basic arithmetic function in OEFs. For a given extension field of order n, n^2 subfield multiplications are required to multiply two values using traditional polynomial multiplication. It is shown in [Knu81] that this can be reduced drastically in certain cases. Using a method developed by Karatsuba and Ofman [KO63], the number of multiplications can be reduced in exchange for an increased number of additions. As long as the time ratio for executing a multiplication vs. an addition is high, this tradeoff is more efficient.

A basic example of Karatsuba is given here to demonstrate its usefulness.

Given two degree-1 polynomials, $A(x)$ and $B(x)$, we can demonstrate the traditional and the Karatsuba methods.

$$\begin{aligned} A(x) &= a_1 x + a_0 \\ B(x) &= b_1 x + b_0 \end{aligned}$$

For the traditional method, we must calculate the product of each possible pair of coefficients.

$$\begin{aligned} D_0 &= a_0 b_0 \\ D_1 &= a_0 b_1 \\ D_2 &= a_1 b_0 \\ D_3 &= a_1 b_1 \end{aligned}$$

Now we can calculate the product $C(x) = A(x) \cdot B(x)$ as:

$$C(x) = D_3 x^2 + (D_2 + D_1)x + D_0$$

The Karatsuba method begins by taking the same two polynomials, and calculating the following three products:

$$\begin{aligned} E_0 &= a_0 b_0 \\ E_1 &= a_1 b_1 \\ E_2 &= (a_0 + a_1)(b_0 + b_1) \end{aligned}$$

These are then used to assemble the result $C(x) = A(x) \cdot B(x)$:

$$C(x) = E_1 x^2 + (E_2 - E_1 - E_0)x + E_0$$

It is easy to verify the results are equal.

We can now look at how many operations are required for each method. The traditional method requires four multiplications and one addition, while the Karatsuba method requires three multiplications and four additions. Thus we have traded a single multiplication for three additions. If the cost to multiply on the target platform is as least three times the cost to add, then the method is effective. While this basic form of Karatsuba was presented in the original paper, there are a number of ways this method may be expanded to handle larger degree polynomials. This is shown in [WP01], where the authors give an in-depth study of this method and its variations.

4.2 Itoh-Tsujii Inversion

Extension field inversion is normally a costly operation, but the nature of OEFs allows the reduction of the extension field inversion to a subfield inversion. The Itoh-Tsujii algorithm [IT88] which was originally developed for use with composite fields $GF(2^{n^m})$ in a normal basis representation can be applied to extension fields $GF(q^m)$ in polynomial representation as shown in [GP01]. It is assumed that the subfield inverse can be calculated by efficient means, such as table-lookup or the Euclidean algorithm, given a small order of the subfield. To perform the OEF inversion, we use the following expression:

$$A^{-1} = (A^r)^{-1} A^{r-1}, \text{ where } r = \frac{q^m - 1}{q - 1}. \tag{4.1}$$

Algorithm 4.1 shows the general case for inversion. It is key to observe that $A^r \in GF(q)$. Since r is known ahead of time, an efficient addition chain for the exponentiation in Step 1 can be precomputed and hardcoded into the algorithm. This can be seen in Algorithms 5.3 and 6.1 where the complete process including the addition chains are shown for the 8-bit and 16-bit fields. In general, an addition chain

Algorithm 4.1 General ITI Algorithm in $GF(q^m)$

Require: $A \in GF(q^m)$

Ensure: $C \equiv A^{-1} \bmod P(x)$

1: $B \leftarrow A^{r-1}$ (using an addition chain)

2: $b \leftarrow BA = A^{r-1}A = A^r \in GF(q)$

3: $b \leftarrow b^{-1} = (A^r)^{-1}$

4: $C \leftarrow bB = (A^r)^{-1}A^{r-1} = A^{-1}$

can be formed utilizing $\lfloor \log_2(m-1) \rfloor + W_H(m-1) - 1$ extension field multiplications, where $W_H(m-1)$ denotes the Hamming weight.

To further reduce the complexity, we utilize the Frobenius map to compute the exponentiations of A occurring in the addition chain. As shown in [BP01], for an OEF with a binomial field polynomial, the pth iteration of the Frobenius map requires at most $m-1$ multiplications in $GF(q)$.

4.3 de Rooij Point Multiplication

As explained in Section 2.1, the primary operation in an elliptic curve cryptosystem is point multiplication, $C = kP$. For large k, computing kP is a costly endeavor. However, well-studied techniques used for ordinary integer exponentiation can be advantageously adapted to this setting. The most basic of these algorithms is the binary-double-and-add algorithm [Knu81]. It has a complexity of $\log_2(k) + W_H(k)$ group operations, where W_H is the Hamming weight of the multiplier k. On average, then, we can expect this algorithm to require $1.5 \log_2(k)$ group operations. Using more advanced methods, such as signed digit, k-ary or sliding window, the complexity may be reduced to approximately $1.2 \log_2(k)$ group operations on average [MvOV97].

The situation is much better in applications where the point is known ahead of time. The most common public-key operation for a smart card or PDA is to provide a digital signature. The ECDSA algorithm [IEE00] involves the multiplication of a fixed curve point by the user-generated private key as the core operation. Because the curve point is known ahead of time, precomputations may be performed to expedite the signing process. Using a method devised by de Rooij in [dR98], we are able to reduce the number of group operations necessary by a factor of four over the binary-double-and-add algorithm. The de Rooij algorithm is a variant of that devised by Brickell, Gordon, McCurley, and Wilson [BGMW93], but requires far fewer precomputations.

A modified form of de Rooij is shown in Algorithm 4.2. Note that the step shown in line 10 requires general point multiplication of A_M by q, where $0 \leq q < b$. This is accomplished using the binary-double-and-add algorithm. In [dR98], the author remarks that during execution, q is rarely greater than 1.

The choice of t and b are very important to the operation of this algorithm. They are defined such that $b^{t+1} \geq \#E(GF(p^m))$. The algorithm must be able to handle a multiplier, s, not exceeding the order of the elliptic curve. The number of point precomputations and temporary storage locations is determined by $t + 1$, while b represents the maximum size of the exponent words. Thus we need to find a compromise between the two parameters. The values chosen for this implementation are covered in Sections 5.4.4 and 6.3.3.

Algorithm 4.2 EC Fixed Point Multiplication using Precomputation and Vector Addition Chains

Require: $\{b^0A, b^1A, \ldots, b^tA\}$, $A \in E(GF(p^m))$, and $s = \sum_{i=0}^{t} s_i b^i$

Ensure: $C = sA$, $C \in E(GF(p^m))$

1: Define $M \in [0, t]$ such that $z_M \geq z_i$ for all $0 \leq i \leq t$

2: Define $N \in [0, t], N \neq M$ such that $z_N \geq z_i$ for all $0 \leq i \leq t, i \neq M$

3: **for** $i \leftarrow 0$ to t **do**

4: $\quad A_i \leftarrow b^i A$

5: $\quad z_i \leftarrow s_i$

6: **end for**

7: Determine M and N for $\{z_0, z_1, \ldots, z_t\}$

8: **while** $z_N \geq 0$ **do**

9: $\quad q \leftarrow \lfloor z_M / z_N \rfloor$

10: $\quad A_N \leftarrow qA_M + A_N$ – general point multiplication

11: $\quad z_M \leftarrow z_M$ **mod** z_N

12: $\quad$ Determine M and N for $\{z_0, z_1, \ldots, z_t\}$

13: **end while**

14: $C \leftarrow z_M A_M$

Chapter 5

8-bit Implementation

5.1 Introduction to the 8051

A typical large-scale smart card application such as retail banking can entail the manufacture, personalization, issuance, and support of millions of smart cards. Due to the grand scale involved, the success of such an application is inherently linked to careful cost management of each of these areas. However, budgetary constraints must be weighed against the basic requirements for smart card security. The security services offered by a smart card often include both data encryption and public-key operations. Creation of a digital signature is often the most computationally intensive operation demanded of a smart card.

Smart cards often use 8-bit microcontrollers derived from 1970s families such as the Intel 8051 [YA95] and the Motorola 6805. The use of public-key algorithms such as RSA or DSA, which are based on modular arithmetic with very long operands, on such a processor predictably results in unacceptably long processing delays. To address this problem, many smart card microcontroller manufacturers include additional on-chip

hardware to accelerate long-number arithmetic operations. However, in cost-sensitive applications it can be attractive to execute public-key operations on smart cards without coprocessors.

5.2 Rough Performance Comparison of Field Types

Field multiplication is the time critical operation in most ECC realizations. To address our need for fast field arithmetic in an ECC implemented on a smart card, we compared three options for finite field arithmetic on a standard Intel 8051 running at 12 MHz. Due to the 8051's internal clock division factor of 12, one internal clock cycle is equivalent to one microsecond. Thus, these timings may be interpreted as either internal clock cycles or microseconds. We implemented extension field multiplication for the three candidates in assembly. We chose a field order of about 2^{135} which provides moderate security as will be discussed in Section 5.3 below. The field elements are represented with a polynomial basis and we took advantage of the standard arithmetic algorithms available for each. For the binary field $GF(2^{135})$ a shift-and-add algorithm was used to emulate a shift register multiplier. It should be noted that faster techniques are available [ITT+99]. For the composite field $GF((2^8)^{17})$ a table look-up is employed to realize subfield multiplication. Results are shown in Table 5.1.

Table 5.1: Extension field multiplication performance on an Intel 8051

Field	*appr. Field Order*	*# Cycles for Multiply*
$GF(2^{135})$	2^{135}	19,600
$GF((2^8)^{17})$	2^{136}	7,479
$GF((2^8-17)^{17})$	2^{134}	5,084

Thus we see that generic binary fields offer performance which lags far behind

the other two options. Further, certain composite fields have recently been shown to have cryptographic weaknesses [GHS00]. This gives further evidence that OEFs are the best choice for our application.

5.3 Remark on the Finite Field Order Chosen

In recent work, Lenstra and Verheul show that under particular assumptions, 990-bit RSA and DSS systems may be considered to be of equivalent security to 135-bit ECC systems [LV00a]. The authors further argue that 135-bit ECC keys are adequate for commercial security in the year 2001. This notion of commercial security is based on the hypothesis that a 56-bit block cipher offered adequate security in 1982 for commercial applications.

The validation of this estimate has more recently been confirmed by the breaking of the ECC2K-108 challenge [HDdRL]. First, note that the field $GF((2^8 - 17)^{17})$ has an order of about 2^{134}. Breaking the Koblitz (or anomalous) curve cryptosystem over $GF(2^{108})$ required slightly more effort than a brute force attack against DES. Hence, an ECC over a 134-bit field which does not use a subfield curve is by a factor of $\sqrt{108} \cdot \sqrt{2^{26}} \approx 2^{16}$ harder to break than the ECC2K-108 challenge or DES. Thus, based on current knowledge of EC attacks, the system proposed here is roughly security equivalent to a 72-bit block cipher. This implies that an attack would require about 65,000 times as much effort as breaking DES. Note also that factoring the 512-bit RSA challenge took only about 2% of the time required to break DES or the ECC2K-108 challenge. This suggests that an ECC over the proposed field $GF(239^{17})$ offers far more security than the 512-bit RSA system which has been popular for fielded smart card applications. In summary, we feel that our selection of field order provides medium-term security which is sufficient for many current smart card

applications.

Of course, the discussion above assumes that there are no special attacks against ECCs over OEFs. This assumption seems to be valid at the time of writing [GHS00].

5.4 Algorithms

When choosing an algorithm to implement on 8-bit processors, it is important that the parameter choices are optimized for the target platform. The Intel 8051 offers a multiply instruction which computes the product of two integers each less than $2^8 = 256$. Thus, we chose a prime $2^8 - 17 = 239$ as our field characteristic so that multiplication of elements in the prime subfield can use the ALU's multiplier. In addition, the nature of the OEF leads to an efficient reduction method. Field elements are represented as polynomials of degree up to 16, with coefficients in the prime subfield $GF(239)$. As mentioned in Section 2.2.4, the polynomial is reduced modulo an irreducible polynomial, $P(x) = x^m - \omega$. In this implementation $P(x) = x^{17} - 2$.

The key performance advantage of OEFs is due to fast modular reduction in the subfield. Given a prime, $p = 2^n - c$, reduction is performed by dividing the number x into two n-bit words. The upper bits of x are "folded" into the lower ones, leading to a very efficient reduction. The central observation is that $2^n \equiv c \bmod 2^n - c$. The basic reduction step which reduces a $2n$-bit value x to a result with $1.5n$ bits is given by representing $x = x_1 2^n + x_0$, where $x_0, x_1 < 2^n$. Thus a reduction is performed by:

$$x \equiv x_1 c + x_0 \bmod 2^n - c, \tag{5.1}$$

which takes one multiplication by c, one addition, and no divisions or inversions. As will be seen in Section 5.4.1, the reduction principle for OEFs is expanded for the implementation described here.

Furthermore, calculating a multiplicative inverse over the 8-bit subfield is easily implemented with table look-up. There is a relative cost in increased codesize, but the subfield inverse requires only two instructions. In contrast, a method such as the Extended Euclidean Algorithm would require a great deal more processing time. This operation is required for our optimized inversion algorithm, as described in Section 5.4.3.

The elliptic curve group operation requires 2 multiplications, 1 squaring, 1 inversion, and a number of additions that are relatively fast compared with the first three. In our case, squaring and inversion performance depends on the speed of multiplication. Therefore the speed of a single extension field multiplication determines the speed of the group operation in general.

Addition is carried out in the extension field by $m-1$ component-wise additions modulo p. Subtraction is performed in a similar manner.

5.4.1 Multiplication

Extension field multiplication is implemented as polynomial multiplication with a reduction modulo the irreducible binomial $P(x) = x^{17} - 2$. This modular reduction is implemented in an analogous manner to the subfield modular reduction outlined above. First, we observe that $x^m \equiv \omega \bmod x^m - \omega$. This observation leads to the general expression for this reduction, given by

$$\begin{aligned} C(x) \equiv\ & c'_{m-1}x^{m-1} + [\omega c'_{2m-2} + c'_{m-2}]x^{m-2} + \cdots \\ & + [\omega c'_{m+1} + c'_1]x + [\omega c'_m + c'_0] \bmod x^m - \omega. \end{aligned} \tag{5.2}$$

Thus, the product C of a multiplication $A \times B$ can be computed as shown in Algorithm 5.1.

As can be seen, extension field multiplication requires m^2 $GF(239)$ products

Algorithm 5.1 Extension Field Multiplication

Require: $A(x) = \sum a_i x^i, B(x) = \sum b_i x^i \in GF(239^{17})/P(x)$, where $P(x) = x^m - \omega$; $a_i, b_i \in GF(239)$; $0 \leq i < 17$

Ensure: $C(x) = \sum c_i x^i = A(x)B(x)$, $c_i \in GF(239)$

First we calculate the intermediate values for c'_i, $i = 17, 18, \ldots, 32$.

$c'_{17} \leftarrow a_1 b_{16} + a_2 b_{15} + \ldots + a_{14} b_3 + a_{15} b_2 + a_{16} b_1$

$c'_{18} \leftarrow a_2 b_{16} + a_3 b_{15} + \ldots + a_{15} b_3 + a_{16} b_2$

$\ldots$

$c'_{31} \leftarrow a_{15} b_{16} + a_{16} b_{15}$

$c'_{32} \leftarrow a_{16} b_{16}$

Now calculate c_i, $i = 0, 1, \ldots, 16$.

$c_0 \leftarrow a_0 b_0 + \omega c'_{17} \bmod 239$

$c_1 \leftarrow a_0 b_1 + a_1 b_0 + \omega c'_{18} \bmod 239$

$\ldots$

$c_{15} \leftarrow a_0 b_{15} + a_1 b_{14} + \ldots + a_{14} b_1 + a_{15} b_0 + \omega c'_{32} \bmod 239$

$c_{16} \leftarrow a_0 b_{16} + a_1 b_{15} + \ldots + a_{14} b_2 + a_{15} b_1 + a_{16} b_0 \bmod 239$

Table 5.2: Inner product maximum value

1. one product multiplication has a maximum value of $(p-1)^2$
2. we accumulate 17 products, 16 of which are multiplied by $\omega = 2$
3. $\text{ACC}_{max} = 33(p-1)^2 = 1869252 = \text{1C85C4h} < 2^{21}$

$a_i b_j$, and $m-1$ multiplications by ω when the schoolbook method for polynomial multiplication is used. These $m^2 + m - 1$ subfield multiplications form the performance critical part of a field multiplication. In the earlier OEF work [Bai98], [BP98], a subfield multiplication was performed as single-precision integer multiplication resulting in a double-precision product with a subsequent reduction modulo p. For OEFs with $p = 2^n \pm c$, $c > 1$, this approach requires 2 integer multiplications and several shifts and adds using Algorithm 14.47 in [MvOV97]. A key idea of this contribution is to deviate from this approach. We propose to perform only one reduction modulo p per coefficient c_i, $i = 0, 1, \ldots, 16$. This is achieved by allowing the residue class of the sum of integer products to be represented by an integer larger than p. The remaining task is to efficiently reduce a result which spreads over more than two words. Hence, we can reduce the number of reductions to m, while still requiring $m^2 + m - 1$ multiplications.

During the product calculations, we perform all required multiplications for a resulting coefficient, accumulate a multi-word integer, and then perform a reduction. The derivation of the maximum value for the multi-word integer c_i before reduction is shown in Table 5.2.

We now expand the basic OEF reduction shown in Equation (5.1) for multiple words. As $\log_2(\text{ACC}_{max}) = 21$ bits, the number can be represented in the radix

Table 5.3: Intermediate reduction maxima

1. Using Equation (5.3), given that $0 \leq x \leq$ 1C85C4h
2. $\mathbf{max}(x') =$ 1734h, when $x =$ 1BFFFFh.
3. Using Equation (5.4), given that $0 \leq x' \leq$ 1734h
4. $\mathbf{max}(x'') =$ 275h, when $x' =$ 16FFh.

2^8 with three digits. We observe $2^n \equiv c \pmod{2^n - c}$ and $2^{2n} \equiv c^2 \pmod{2^n - c}$. Thus the expanded reduction for operands of this size is performed by representing $x = x_2 2^{2n} + x_1 2^n + x_0$, where $x_0, x_1, x_2 < 2^n$. The first reduction is performed as

$$x' \equiv x_2 c^2 + x_1 c + x_0 \pmod{2^n - c}, \tag{5.3}$$

noting that $c^2 = 289 \equiv 50 \bmod 239$. The reduction is repeated, now representing the previous result as $x' = x'_1 2^n + x'_0$, where $x'_0, x'_1 < 2^n$. The second reduction is performed as

$$x'' \equiv x'_1 c + x'_0 \bmod 2^n - c. \tag{5.4}$$

The maximum intermediate values through the reduction are shown in Table 5.3. Line 1 shows the maximum sum after inner product addition. While this value is the largest number that will be reduced, it is more important to find the maximum value that can result from the reduction. This case can be found by maximizing x_1 and x_0 at the cost of reducing x_2 by one. Looking at Table 5.3 again, this value is shown in line 2, as is the resulting reduced value. The process is repeated again in lines 3 and 4, giving us the maximum reduced value after two reductions.

Note that through two reductions, we reduced a 21-bit input to 13 bits, and

finally to 10 bits. At this point in the reduction, we could perform the same reduction again, but it would only provide a slight improvement. Adding $x_1''c + x_0''$ would result in a 9-bit number. Therefore it is more efficient to write custom code to handle various possibilities. Most important is to eliminate the two highest order bits, and then to ensure the resulting 8-bit number is the least positive representative of its residue class. The entire multiplication and reduction is shown in Algorithm 5.2.

To perform the three-word reduction requires three 8-bit multiplications and then several comparative steps. After the first two multiplications, the inner product sum has been reduced to a 13-bit number. If we were to reduce each inner product individually, every step starting at line 13 in Algorithm 5.2 would be required. Ignoring the trailing logic, which would add quite a bit of time itself, this would require $m = 17$ multiplications as opposed to the three needed in Algorithm 5.2. By accumulating inner products and performing a single reduction we have saved 14 multiplications, plus additional time in trailing logic, per coefficient calculation.

5.4.2 Squaring

Extension field squaring is similar to multiplication, except that the two inputs are equal. By modifying the standard multiplication routine, we are able to take advantage of identical inner product terms. For example, $c_2 = a_0b_2 + a_1b_1 + a_2b_0 + \omega c_{19}$, can be simplified to $c_2 = 2a_0a_2 + {a_1}^2 + \omega c_{19}$. Further gain is accomplished by doubling only one coefficient, reducing it, and storing the new value. This approach saves us from recalculating the doubled coefficient when it is needed again.

Algorithm 5.2 Extension Field Multiplication with Subfield Reduction

Require: $A(x) = \sum a_i x^i, B(x) = \sum b_i x^i \in GF(239^{17})/P(x)$, where $P(x) = x^m - \omega;\ a_i, b_i \in GF(239);\ 0 \leq i < 17$

Ensure: $C(x) = \sum c_i x^i = A(x)B(x),\ c_i \in GF(239)$

1: Define $z[w]$ to mean the w-th 8-bit word of z
2: $c_i \leftarrow 0$
3: **if** $i \neq 16$ **then**
4: **for** $j \leftarrow m-1$ downto $i+1$ **do**
5: $c_i \leftarrow c_i + a_{i+m-j} b_j$
6: **end for**
7: $c_i \leftarrow 2c_i$ – multiply by $\omega = 2$
8: **end if**
9: **for** $j \leftarrow i$ downto 0 **do**
10: $c_i \leftarrow c_i + a_{i-j} b_j$
11: **end for**
12: $c_i \leftarrow c_i[2] * 50 + c_i[1] * 17 + c_i[0]$ – begin reduction, Equation (5.3)
13: $t \leftarrow c_i[1] * 17$ – begin Equation (5.4)
14: **if** $t \geq 256$ **then**
15: $t \leftarrow t[0] + 17$
16: **end if**
17: $c_i \leftarrow c_i[0] + t$ – end Equation (5.4)
18: **if** $c_i \geq 256$ **then**
19: $c_i \leftarrow c_i[0] + 17$
20: **if** $c_i \geq 256$ **then**
21: $c_i \leftarrow c_i[0] + 17$
22: **terminate**
23: **end if**
24: **end if**
25: $c_i \leftarrow c_i - 239$
26: **if** $c_i \leq 0$ **then**
27: $c_i \leftarrow c_i + 239$
28: **end if**

5.4.3 Inversion

Inversion in the OEF is performed via a modification of the Itoh-Tsujii algorithm [IT88] as shown in [GP01], which reduces the problem of extension field inversion to subfield inversion. The algorithm computes an inverse in $GF(p^{17})$ as $A^{-1} = (A^r)^{-1}A^{r-1}$ where $r = (p^{17} - 1)/(p - 1) = 11 \ldots 10_p$. Algorithm 5.3 shows the details of this method. A key point is that $A^r \in GF(p)$ and is therefore an 8-bit value [LN83]. Therefore the step shown in line 10 is only a partial extension field multiplication, as all coefficients of A^r other than b_0 are zero. Inversion of A^r in the 8-bit subfield is performed via table look-up.

The most costly operation is the computation of A^r. Because the exponent is fixed, an addition chain can be derived to perform the exponentiation. For $m = 17$, the addition chain requires 4 multiplications and 5 exponentiations to a p^i-th power. The element is then inverted in the subfield, and then multiplied back in as shown in steps 11 and 12 of Algorithm 5.3. This operation results in the multiplicative inverse A^{-1}.

The Frobenius map raises a field element to the p-th power. In practice, this automorphism is evaluated in an OEF by multiplying each coefficient of the element's polynomial representation by a "Frobenius constant," determined by the field and its irreducible binomial. A list of the constants used is shown in Table 5.4. To raise a given field element to the p^i-th power, each a_j, $j = 0, 1, \ldots, 16$, coefficient are multiplied by the corresponding constant in the subfield $GF(239)$.

Thus we have efficient methods for both the exponentiation and subfield inversion required in Algorithm 5.3. Our results in Section 5.6 show the ratio of extension field multiplication time to extension field inversion time is 1:4.8. This ratio indicates that an affine representation of the curve points offers better performance than the corresponding projective-space approach, which eliminates the need for an inversion

Table 5.4: Frobenius constants $B(x) = A(x)^{p^i}$

	Exponent			
Coefficient	p	p^2	p^4	p^8
a_0	1	1	1	1
a_1	132	216	51	211
a_2	216	51	211	67
a_3	71	22	6	36
a_4	51	211	67	187
a_5	40	166	71	22
a_6	22	6	36	101
a_7	36	101	163	40
a_8	211	67	187	75
a_9	128	132	216	51
a_{10}	166	71	22	6
a_{11}	163	40	166	71
a_{12}	6	36	101	163
a_{13}	75	128	132	216
a_{14}	101	163	40	166
a_{15}	187	75	128	132
a_{16}	67	187	75	128

Algorithm 5.3 Inversion Algorithm in $GF((2^8 - 17)^{17})$

Require: $A \in GF(p^{17})$

Ensure: $B \equiv A^{-1} \bmod P(x)$

1: $B_0 \leftarrow A^p = A^{(10)_p}$

2: $B_1 \leftarrow B_0 A = A^{(11)_p}$

3: $B_2 \leftarrow (B_1)^{p^2} = A^{(1100)_p}$

4: $B_3 \leftarrow B_2 B_1 = A^{(1111)_p}$

5: $B_4 \leftarrow (B_3)^{p^4} = A^{(11110000)_p}$

6: $B_5 \leftarrow B_4 B_3 = A^{(11111111)_p}$

7: $B_6 \leftarrow (B_5)^{p^8} = A^{(1111111100000000)_p}$

8: $B_7 \leftarrow B_6 B_5 = A^{(1111111111111111)_p}$

9: $B_8 \leftarrow (B_7)^p = A^{(1111111111111110)_p}$

10: $b \leftarrow B_8 A = A^{r-1} A = A^r$

11: $b \leftarrow b^{-1} = (A^r)^{-1}$

12: $B \leftarrow b B_8 = (A^r)^{-1} A^{r-1} = A^{-1}$

in every group operation at the expense of many more multiplications.

5.4.4 Point Multiplication

As discussed in Section 4.3, the algorithm developed by de Rooij is used for point multiplication. The parameter choices for this algorithm are crucial to balance storage requirements and speed gains. Two obvious choices for an 8-bit architecture are $b = 2^{16}$ and $b = 2^8$, since dividing the exponent into radix b words is essentially free as they align with the memory structure. This results in a precomputation count of 9 and 18 points, respectively. The tradeoff here is the cost of memory access vs. arithmetic speeds. As we double the number of precomputed points, the algorithm operates only marginally faster, as shown in [dR98], but the arithmetic operations are easier to perform on the 8-bit microcontroller. Moreover, as the number of precomputed points grows, the cost to access the memory in which the points are stored grows, outweighing the benefits of further precomputation. Even though the XRAM may be physically internal to the microcontroller, it is outside the natural address space, and thus is more expensive to access than internal RAM.

For $b = 2^{16}$, we must perform 16-bit multiplication and modular reduction, but only need to store 9 precomputed points and 9 temporary points. For $b = 2^8$, however, we must now store 18 precomputed points and 18 temporary points, but now only have to perform 8-bit multiplication and modular reduction. Implementation results show that the speed gain from doubling the precomputations and the faster 8-bit arithmetic slightly outweighs the cost of the increase in data access, as shown in Section 5.6, assuming a microcontroller with enough XRAM is available.

5.5 Implementation Details

Implementing ECCs on the 8051 is a challenging task. The processor has only 256 bytes of internal RAM, and only the lower 128 bytes are directly addressable. The upper 128 bytes must be referenced through the use of the two pointer registers: R0 and R1. Accessing this upper storage takes more time per operation and incurs more overhead in manipulating the pointers. To make matters worse, the lower half of the internal RAM must be shared with the system registers and the stack, thus leaving fewer memory locations free. XRAM may be utilized, but there is essentially only a single pointer for these operations, which are typically at least three times slower than their internal counterparts.

This configuration makes the 8051 a tight fit for an ECC. Each curve point in our group occupies 34 bytes of RAM, 17 bytes each for the X and Y coordinates. To make the system as fast as possible, the most intensive field operations, such as multiplication, squaring, and inversion, operate on fixed memory addresses in the faster, lower half of RAM. During a group operation, the upper 128 bytes are divided into three sections for the two input and one output curve points, while the available lower half of RAM is used as a working area for the field arithmetic algorithms. A total of four 17-byte coordinate locations are used, starting from address 3Ch to 7Fh, the top of lower RAM. This is illustrated in Table 5.5.

Finally, six bytes, located from 36h to 3Bh, are used to keep track of the curve points, storing the locations of each curve point in the upper RAM. Using these pointers, we can optimize algorithms that must repeatedly call the group operation, often using the output of the previous step as an input to the next step. Instead of copying a resulting curve point from the output location to an input location, which involves using pointers to move 34 bytes around in upper RAM, we can simply change the pointer values and effectively reverse the inputs and outputs of the group

Table 5.5: Internal RAM memory allocation

Address	*Function*
00–07h	Registers
08–14h	de Rooij Algorithm Variables
15–35h	Call Stack (variable size)
36–3Bh	Pointers to Curve Points in Upper RAM
3C–7Fh	Temporary Field Element Storage
80–E5h	Temporary Curve Point Storage
E6–FFh	Unused

operation.

The arithmetic components are all implemented in handwritten, loop-unrolled assembly. This results in large, but fast and efficient program code, as shown in Table 5.7. Note that the execution times are nearly identical to the code size, an indication of their linear nature. Each arithmetic component is written with a clearly defined interface, making them completely modular. Thus, a single copy of each component exists in the final program, as each routine is called repeatedly.

Extension field inversion is constructed using a number of calls to the other arithmetic routines. The group operation is similarly constructed, albeit with some extra code for point equality and inverse testing. The binary-double-and-add and de Rooij algorithms were implemented in C, making calls to the group operation assembly code when needed. Looping structures were used in both programs as the overhead incurred is not as significant as it would be inside the group operation and field arithmetic routines. The final size and architecture requirements for the programs are shown in Table 5.6.

Table 5.6: Program size and architecture requirements

Type	Size (bytes)	Function
Code	13K	Program Storage
Internal RAM	183	Finite Field Arithmetic
External RAM	306	Temporary Points
	34	Integer Multiplicand
Fixed Storage	306	Precomputed Points

5.6 Results

Our target microcontroller is the Infineon SLE44C24S, an 8051 derivative with 26 kilobytes of ROM, 2 kilobytes of EEPROM, and 512 bytes of XRAM. This XRAM is in addition to the internal 256 bytes of RAM, and its use incurs a much greater delay, as noted in Section 5.4.4. However, this extra memory is crucial to the operation of the de Rooij algorithm which requires the manipulation of several precomputed curve points.

The Keil PK51 tools were used to assemble, debug and time the algorithms, since we did not have access to a simulator for the Infineon smart card microcontrollers. Thus, to perform timing analysis a generic Intel 8051 was used, running at 12 MHz. Given the optimized architecture of the Infineon controller, an SLE44C24S running at 5 MHz is roughly speed equivalent to a 12 MHz Intel 8051.

Table 5.7: Finite field arithmetic performance on a 12 MHz 8051

Description	Operation	Time [a] (μsec)	Code Size (bytes)
Multiplication	$C(x) = A(x)B(x)$	5084	5110
Squaring	$C(x) = A(x)^2$	3138	3259
Addition	$C(x) = A(x) + B(x)$	266	360
Subtraction	$C(x) = A(x) - B(x)$	230	256
Inversion	$C(x) = A(x)^{-1}$	24489	[b]
Scalar Mult.	$C(x) = sA(x)$	642	666
Scalar Mult. by 2	$C(x) = 2A(x)$	180	257
Scalar Mult. by 3	$C(x) = 3A(x)$	394	412
Frobenius Map	$C(x) = A(x)^{p^i}$	625	886
Partial Multiplication	c_0 of $A(x)B(x)$	303	305
Subfield Inverse	$c = a^{-1}$	4	236

[a] Time calculated averaging over at least 5,000 executions with random inputs

[b] Inversion is a collection of calls to the other routines and has negligible size itself.

Using each of the arithmetic routines listed in Table 5.7, the elliptic curve group operation takes 39.558 msec per addition and 43.025 msec per doubling on average on the 12 MHz Intel 8051.

Using random exponents, we achieve a speed of 8.37 seconds for point multiplication using binary-double-and-add. This is exactly what would be predicted given the speed of point addition and doubling. If we fix the curve point and use the de Rooij algorithm discussed in Section 4.3, we achieve speeds of 1.95 seconds and 1.83 seconds, for 9 and 18 precomputations, respectively. This is a speed-up factor of well over four when compared to general point multiplication. Unfortunately, our target

Table 5.8: Elliptic curve performance on a 12 MHz 8051

Operation	Method	Time (msec)
Point Addition		39.558
Point Double		43.025
Point Multiplication	Binary Method	8370
Point Multiplication	de Rooij w/9 precomp.	1950
Point Multiplication	de Rooij w/18 precomp.	1830

microcontroller, the SLE44C24S, only has 512 bytes of XRAM where we manipulate our precomputed points. Since we require 34 bytes per precomputed point, 18 temporary points will not fit in the XRAM, limiting us to 9 temporary points on this microcontroller. These results are summarized in Table 5.8.

If we were to expand our focus beyond smart cards, we could choose one of the many 8051 derivatives. For example, Dallas Semiconductor offers an 8051 designed for security applications that can be clocked as fast as 33 MHz [Dal]. Furthermore, this processor has an enhanced core similar to the Infineon processor that was the focus of this implementation. In total, this chip could execute a point multiplication over 6 times faster than the results above. This implementation would result in a fixed point multiplication in under 400 msec on a chip that is pin for pin compatible with an 8051. It is accepted that a 33 MHz clock is not found in a typical 8051, but it is mentioned to underscore that strong cryptographic operations are possible in these modest processors using these techniques.

Chapter 6

16-bit Implementation

6.1 Introduction to the MC68328

The second implementation is targeted towards the pervasive Palm computing platform. These devices dominate the handheld organizer field through a combination of low cost and long battery life. A pair of AAA batteries can typically power one of these devices for several weeks due to aggressive power management built around a low power 16-bit processor. In typical applications, the processor is in a sleep mode over 99% of the time, awakening only when the user prompts it into action via the touch screen or other button.

While these devices have many communication capabilities, they are completely lacking in security. All user data, including those marked "private" are stored in the clear, protected only with trivial password-based methods [Ats00]. The password is stored on the Palm and transmitted to the host PC encoded using a simple reversible function. Therefore, access to either the Palm or PC results in the knowledge of both the "private" data and the password itself. The absence of strong

public-key security is a direct result of the lack of CPU (and battery) power. Utilizing traditional algorithms would incur unacceptable delays and increase the battery drain.

The specific processor behind the Palm is the Motorola Dragonball, an updated low-power member of the 68000 family. These embedded CPUs operate at clock frequencies between 15 and 25 MHz, but, as stated earlier, achieve long battery life by sleeping for a majority of the time.

It is important to note that while a specific processor was chosen here, the implementation is optimized solely for its specific word size. Thus this research is applicable to any 16-bit processor.

6.2 Field Order

Because the processor has a native 16-bit word size, and thus a 16-bit integer multiplier, we should choose an OEF with a subfield of this size. While many pseudo-Mersenne primes exist in this range, we also need to keep in mind the size of the extension degree and the existence of a suitable irreducible binomial. A list of possible fields can be found in the appendix of [BP01]. Because the Dragonball provides us with a bit more processing power than the 8051, we can choose a larger field order that would provide a degree of security comparable to RSA 1024. An OEF that meets all these requirements is $GF((2^{13}-1)^{13})$ with the irreducible binomial $P(x) = x^{13}-2$. This construction yields a field order of about 2^{169}. In [LV00a], the authors argue that 169-bit ECC keys are adequate for commercial security until the year 2013.

6.3 Algorithms

The implementation of an ECC over $GF((2^{13}-1)^{13})$ closely follows the methods described in the previous chapter. For that reason, only the algorithmic differences between the two will be highlighted here.

Subfield reduction is handled differently in this implementation than it is in Section 5.4 for the 8-bit microcontroller. As can be seen in [Mot93], the MC68328 supports a native 32-bit divide instruction which will be utilized instead of the normal OEF subfield reduction. More about this will be discussed in Section 6.4. Subfield inversion is performed using a look-up table.

6.3.1 Multiplication

Like the 8-bit implementation, extension field multiplication is implemented as polynomial multiplication with a reduction modulo the irreducible binomial $P(x) = x^{13} - 2$. The modular reduction is implemented using the observation that $x^m \equiv \omega \bmod x^m - \omega$. Thus the product C is computed using the same general structure as Algorithm 5.1 in Chapter 5 but with the specific constants applicable for $GF((2^{13}-1)^{13})$, substituted appropriately.

During the calculation of each coefficient c_k of C, the inner products $a_i b_j$ are allowed to accumulate producing the maxima shown in Table 6.1 after extension field reduction, but before any subfield reduction.

The MC68328 microprocessor supports a native division operation:
32-bit dividend / 16-bit divisor $\rightarrow$ 16-bit quotient and 16-bit remainder [Mot93]. Thus, this is a special case where modular reduction may be performed directly. The performance implications of this choice are discussed in Section 6.4. If such an

Table 6.1: Inner product maximum value – 16-bit case

1. one subfield multiplication with a maximum value of $(p-1)^2 = 67076100$
2. we accumulate 13 products, 12 of which are multiplied by $\omega = 2$
3. $\text{ACC}_{max} = 25(p-1)^2 = 1676902500 < 2^{32}$

operation were not available in the processor, the normal OEF subfield reduction would be employed.

A second variant of the extension field multiplication uses the Karatsuba method introduced in Chapter 4. The Karatsuba algorithm calculates a polynomial multiplication trading inner products for more additions and subtractions. Care must be taken not to have any negative intermediate sums as these would not be reduced correctly modulo p. The solution is to perform a subtraction by x as an addition of $p - x$.

A single iteration of Karatsuba is implemented by splitting the field elements A and B:

$$A(x) = A_H x^7 + A_L$$
$$B(x) = B_H x^7 + B_L$$

These four polynomials are used to calculate the following products:

$$\begin{aligned} D_0(x) &= A_L(x) \times B_L(x) \\ D_1(x) &= A_H(x) \times B_H(x) \\ D_{0,1}(x) &= (A_H(x) + A_L(x)) \times (B_H(x) + B_L(x)) \end{aligned}$$

The product $C(x) = A(x) \times B(x)$ is finally calculated by:

$$C(x) \quad = \quad D_1(x)x^{14} + (D_{0,1}(x) - D_1(x) - D_0(x))x^7 + D_0(x)$$

Compared to the schoolbook multiplication method which requires $m^2 = 169$ inner products, this method requires $6^2 + 2(7^2) = 134$ inner products.

6.3.2 Inversion

The addition chain for ITI inversion in $GF((2^{13} - 1)^{13})$ is shown in Algorithm 6.1.

Algorithm 6.1 Inversion Algorithm in $GF((2^{13} - 1)^{13})$

Require: $A \in GF(p^{13})$

Ensure: $B \equiv A^{-1} \bmod P(x)$

1: $B_0 \leftarrow A^p = A^{(10)_p}$

2: $B_1 \leftarrow B_0 A = A^{(11)_p}$

3: $B_2 \leftarrow (B_1)^{p^2} = A^{(1100)_p}$

4: $B_3 \leftarrow B_2 B_1 = A^{(1111)_p}$

5: $B_4 \leftarrow (B_3)^{p^4} = A^{(11110000)_p}$

6: $B_5 \leftarrow B_4 B_3 = A^{(11111111)_p}$

7: $B_6 \leftarrow (B_5)^{p^4} = A^{(111111110000)_p}$

8: $B_7 \leftarrow B_6 B_3 = A^{(111111111111)_p}$

9: $B_8 \leftarrow (B_7)^p = A^{(1111111111110)_p}$

10: $b \leftarrow B_8 A = A^{r-1} A = A^r$

11: $b \leftarrow b^{-1} = (A^r)^{-1}$

12: $B \leftarrow b B_8 = (A^r)^{-1} A^{r-1} = A^{-1}$

The cost of the algorithm can be analyzed easily. The addition chain to calculate A^r requires 4 extension field multiplications and 5 exponentiations to a p^i-th

power. Then a partial multiplication and a scalar multiplication are required to complete the inverse. The subfield inverse in step 11 is completed via table look-up as will be discussed in Section 6.4.

6.3.3 Point Multiplication

As with the 8-bit field, the method developed by de Rooij is used to speed-up point multiplication. This method uses an addition chain and a number of precomputed points. As mentioned in Section 5.4.4 the number of precomputed points is a time/space tradeoff. Each precomputed point occupies 52 bytes of memory. Furthermore, during execution, additional temporary space is required. This is detailed in Section 6.4.

For the MC68328, three possible base sizes are likely: $b = 2^{32}$, $b = 2^{16}$, and $b = 2^8$. These would result in 6, 11, and 22 precomputations and speed-up factors of 3.4, 4 and 4.6 respectively [dR98]. Outside of these base sizes the algorithm benefits increase marginally. Furthermore, the cost of handling a base smaller than 8 bits would incur additional costs as the natural byte boundaries can not be used to divide the exponent.

6.4 Implementation Details

For purposes of comparison, three versions of the extension field multiplication algorithm are implemented in C. First, the schoolbook method is implemented using two nested loops. Second, the same algorithm is completely unrolled by providing code to calculate each inner product. This eliminates the logic surrounding the loop structures and allows the memory offsets to be fixed at design time. Since the target

platform possesses ample storage for code and data, the relative code size of each of these implementations is less important than run-time performance.

One notable exception is the use of a look-up table for subfield inversion. This table contains $p - 1$ entries, each 16 bits, resulting in over 16 kilobytes of space for inversion. If space requirements are tight, then the extended binary Euclidean algorithm is used to calculate the subfield inverses at an increased run-time cost, but without the need for a large look-up table.

Finally, the Karatsuba method as described in Section 6.3.1 with one iteration is implemented. First the three D polynomials are generated and, from them, C is calculated. This implementation requires more memory than the previous algorithms as the coefficients of D must be stored.

As mentioned in Section 6.3 the processor supports a 32-bit division instruction. Because in this special case the inner product accumulation does not exceed the 32-bit double word as shown in Section 6.3.1, the processor's native divide instruction can be utilized to compute the reduction efficiently. If the code is written in assembly, the special OEF reduction should be slightly faster than the divide instruction. Combined with the possibility of a faster Karatsuba realization, this suggests further work in that area.

As stated in Section 6.3.3 the de Rooij method requires 52 bytes of storage per precomputed point. During execution of the algorithm, another 52 bytes of RAM per precomputed point are required for temporary storage. For the two most promising bases $b = 2^{16}$ and $b = 2^8$, the storage requirements, in bytes, are 594 permanent, 702 temporary for $b = 2^{16}$ and 1188 permanent and 1296 temporary for $b = 2^8$. Since 2.5 kilobytes of RAM is not excessive for a Palm application, it is recommended to use $b = 2^8$. If multiple points are to be stored however, it might be reasonable to use a larger basis to save on storage.

6.5 Results

The development environment used for the Palm was the Metrowerks Codewarrior for Palm OS. This environment allows for the full simulation, debugging and timing of programs on the actual device. The timings for extension field multiplication are shown in Table 6.2. These results were measured using a Palm Vx, which contains an MC68328EZ running at 20 MHz.

Table 6.2: Finite field multiplication performance on a 20 MHz MC68328EZ

Description	Time (msec) [a]
Looped C	2.41
Unrolled C	1.32
Karatsuba	2.34

[a] Time calculated averaging over at least 10,000 executions with random inputs

Since the most expensive operation in an ECC is extension field multiplication, only this core routine is implemented on the target platform. We note that by using the Itoh-Tsujii Algorithm for inversion, the inversion cost is dominated by multiplication. From this metric, it is trivial to calculate the cost of squarings and inversion as multiples of multiplication. Furthermore, the experimental results from the full 8-bit implementation on the 8051 verify our estimations as to the complexity of the group operation and, in turn, point multiplication. Therefore it is important to note that the following results are estimated based on the calculated complexity.

Implementation results from Section 5.6 show that the ratio of extension field multiplication to inversion is 1:4.8 for the 8-bit implementation. This matches with the anticipated value as inversion requires 4 full multiplications, and a number of extra calculations which have a cost comparable to scalar multiplication. By comparing

Algorithms 5.3 and 6.1, we can observe the complexity of extension field inversion for the 8-bit field is identical to the complexity of extension field inversion for the 16-bit field, in terms of finite field operations. Thus we will use the same ratio of 1:4.8 for multiplication vs. inversion in the 16-bit field. If the extended binary Euclidean algorithm is used for subfield inversion instead of a look-up table, the ratio would have to be increased appropriately.

The group operation complexities are exactly the same as in the 8-bit field for point addition and doubling. The time-critical operations in an addition are an inversion, two multiplications and a squaring. Adding in the extra functions, this results in a ratio of 8 extension field multiplications to a point addition. A point doubling requires an extra squaring and two scalar multiplications resulting in a ratio of approximately 9 extension field multiplications to a point doubling.

Using these ratios and the timing results in Table 6.2 for the fastest multiplication implementation, a time of 10.6 msec and 11.9 msec is estimated for point addition and doubling, respectively.

Using random exponents and the binary-double-and-add algorithm, we can expect an average time of 2.9 seconds for a point multiplication. If the point is fixed, we can take advantage of precomputation and the algorithm developed by de Rooij as discussed in Section 4.3. With 22 precomputations, a point multiplication takes an estimated 630 milliseconds. These results are summarized in Table 6.3.

It should be noted that additional performance gains could be obtained by optimizing the Karatsuba implementation. Using unrolled optimized assembly to implement recursive Karatsuba [WP01] should result in the most speed efficient design using the techniques presented here.

Table 6.3: Estimated elliptic curve performance on a 20 MHz MC68328EZ

Operation	Method	Time (msec)
Point Addition		10.6
Point Double		11.9
Point Multiplication	Binary Method	2900
Point Multiplication	de Rooij w/11 precomp.	725
Point Multiplication	de Rooij w/22 precomp.	630

Chapter 7

Discussion

7.1 Summary

We have demonstrated that a scalar multiplication of a fixed point of an EC can be performed in under 2 seconds on an 8051 microcontroller, in a field of order approximately 2^{134}. This is the core operation for signature generation in the ECDSA scheme. Although the performance and security threshold may not allow the use of our implementation in all smart card applications, we believe that there are scenarios where these parameters offer an attractive alternative to more costly smart cards with coprocessors, especially if public-key capabilities are added to existing systems. We also believe that this implementation can be further improved. In practice, a smart card with an 8051-derived microcontroller that can be clocked faster than the 5 MHz assumed in Section 5.6 can potentially yield point multiplication times which are below one second.

The 16-bit implementation demonstrated how these techniques can be extended to a more powerful embedded microcontroller. The crucial step is choosing

the OEF that best meets the design criteria for field order and computational complexity. We estimate an ECDSA signature operation in under 700 milliseconds on a Palm handheld organizer in a field of order approximately 2^{169}. This performance and security threshold allow its deployment in a large variety of applications. In addition, this field could be implemented efficiently in a 16-bit smart card microcontroller such as the Infineon SLE66C80S to provide a more secure implementation than the 8-bit field. In both of these cases, designing the core implementation routines in assembly would allow for the full speed gains to be realized. Further, the use of an elliptic curve defined over the prime subfield, as suggested in [KMKH99], could also provide a speedup. Each of these potential improvements provides further possibilities to apply the fast field arithmetic provided by an OEF to construct ECCs on embedded microcontrollers without additional coprocessors.

7.2 Conclusions

The following insights are the main achievements of this research:

- We have found that OEFs are good arithmetic structures to use as the underlying finite field in ECCs on embedded microprocessors.
- Postponing the subfield modular reduction during multiplication is an efficient practice on the embedded μPs investigated.
- A careful choice of the underlying finite field, and a combination of optimization techniques, can result in acceptable performance of ECCs on the μPs investigated without the need for specialized coprocessors.

Chapter 8

Future Research

In this chapter we will present an overview of the opportunities for continuation of the work in this thesis. The following ideas came about during this research but were beyond the scope of this work.

8.1 Optimized 16-bit Implementation

The implementation of extension field multiplication for the 16-bit field was written entirely in the standard C language. While this makes the implementation portable, it often does not result in the most efficient code. Most notably, the Karatsuba implementation seemed to be most affected by this design choice. Once a target platform is chosen, an optimized assembly implementation might yield a significant improvement. Given the direct relation of extension field multiplication performance to the performance of the ECC as a whole, this additional effort would be justified. Also, it would be worthwhile comparing, in detail, the exploitation of the pseudo-Mersenne primes for subfield modulo reduction to the division used in Section 6.4.

8.2 Elliptic Curve Enhancements

Several possibilities exist relating to the elliptic curve component of the implementation. First, OEFs can be implemented over Koblitz curves [KMKH99], simplifying the point multiplication operation. Alternatively, we could utilize Montgomery's EC multiplication method to enhance the group operation speed. Briefly, this method calculates only the x coordinate in the EC group operation and later determines the corresponding y coordinate.

Furthermore, this work could be continued by comparing these results to implementations of the curves proposed by NIST in Appendix 6 of [NIS00] on similar 8-bit and 16-bit processors.

8.3 Discrete Log Cryptosystems over OEFs

Finally, the application of OEFs is not limited to ECCs. If we extend our scope, we can apply OEFs to systems based on the discrete log problem as described in [IEE01]. A similar approach is presented in [LV00b]. This would allow the efficient implementation of these algorithms using the techniques presented in this work. Fields of much higher order would be required, (approximately 1024–2048 bits long) but the techniques for fast extension field multiplication and inversion would remain valid. This would enhance the performance of these systems and make their efficient implementation possible on smaller devices.

Bibliography

[Ats00] Atstake. PalmOS Password Retrieval and Decoding. Technical report, 2000.
`http://www.atstake.com/research/advisories/2000/a092600-1.txt`

[Bai98] D. V. Bailey. Optimal Extension Fields. Major Qualifying Project (Senior Thesis), 1998. Computer Science Department, Worcester Polytechnic Institute, Worcester, MA, USA.

[BGMW93] E. F. Brickell, D. M. Gordon, K. S. McCurley, and D. B. Wilson. Fast exponentiation with precomputation. In *Advances in Cryptography — EUROCRYPT '92*, pages 200–207. Springer-Verlag, 1993.

[BP98] D. V. Bailey and C. Paar. Optimal Extension Fields for Fast Arithmetic in Public-Key Algorithms. In *Advances in Cryptology – CRYPTO '98*. Springer-Verlag Lecture Notes in Computer Science, 1998.

[BP01] D. V. Bailey and C. Paar. Efficient Arithmetic in Finite Field Extensions with Application in Elliptic Curve Cryptography. *Journal of Cryptology*, 14(3):153–176, 2001.

[BSS99] I. Blake, G. Seroussi, and N. Smart. *Elliptic Curves in Cryptography*. Cambridge University Press, 1999.

[Cer98] Certicom Corp. The Elliptic Curve Cryptosystem for Smart Cards. online white paper, 1998.
`http://www.certicom.com/research/wecc4.html`

[Dal] Dallas Semiconductor Inc. DS80C310 High-Speed Micro. World Wide Web.
`http://pdfserv.maxim-ic.com/arpdf/DS80C310.pdf`

[DBV+96] E. De Win, A. Bosselaers, S. Vandenberghe, P. De Gersem, and J. Vandewalle. A fast software implementation for arithmetic operations in $GF(2^n)$. In *Asiacrypt '96*. Springer-Verlag Lecture Notes in Computer Science, 1996.

[DMPW98] E. De Win, S. Mister, B. Preneel, and M. Wiener. On the Performance of Signature Schemes Based on Elliptic Curves. In *Algorithmic Number Theory: Third International Symposium*, pages 252–266, Berlin, 1998. Springer-Verlag Lecture Notes in Computer Science.

[dR98] P. de Rooij. Efficient exponentiation using precomputation and vector addition chains. In *Advances in Cryptography — EUROCRYPT '98*, pages 389–399. Springer-Verlag, 1998.

[EGH00] D. Estrin, R. Govindan, and J. Heidemann. Embedding the internet: Introduction. *Communications of the ACM*, 43(5):38–42, May 2000.

[GHS00] P. Gaudry, F. Hess, and N. P. Smart. Constructive and Destructive Facets of Weil Descent on Elliptic Curves. technical report HPL 2000-10, 2000.
`http://www.hpl.hp.com/techreports/2000/HPL-2000-10.html`

[GP97] J. Guajardo and C. Paar. Efficient Algorithms for Elliptic Curve Cryptosystems. In *Advances in Cryptology — Crypto '97*, pages 342–356. Springer-Verlag Lecture Notes in Computer Science, August 1997.

[GP01] J. Guajardo and C. Paar. Itoh-Tsujii Inversion in Standard Basis and Its Application in Cryptography and Codes. *Design, Codes, and Cryptography*, 2001. to appear.

[HDdRL] R. Harley, D. Doligez, D. de Rauglaudre, and X. Leroy. World Wide Web.
`http://cristal.inria.fr/~harley/ecdl7/`

[IEE00] IEEE. Standard Specifications for Public Key Cryptography, IEEE P1363-2000 Standard, 2000.

[IEE01] IEEE. Standard Specifications for Public Key Cryptography: Additional Techniques, IEEE P1363a Standard, 2001. working document.

[IT88] T. Itoh and S. Tsujii. A fast algorithm for computing multiplicative inverses in $GF(2^m)$ using normal bases. *Information and Computation*, 78:171–177, 1988.

[ITT+99] K. Itoh, M. Takenaka, N. Torii, S. Temma, and Y. Kurihara. Fast Implementation of Public-Key Cryptography on a DSP TMS320C6201. In *CHES '99*, pages 61–72. Springer-Verlag, 1999.

[KMKH99] T. Kobayashi, H. Morita, K. Kobayashi, and F. Hoshino. Fast Elliptic Curve Algorithm Combining Frobenius Map and Table Reference to Adapt to Higher Characteristic. In *Advances in Cryptography — EUROCRYPT '99*. Springer-Verlag Lecture Notes in Computer Science, 1999.

[Knu81] D. E. Knuth. *The Art of Computer Programming. Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, Massachusetts, 2nd edition, 1981.

[KO63] A. Karatsuba and Y. Ofman. Multiplication of multidigit numbers on automata. *Sov. Phys.-Dokl. (Engl. transl.)*, 7(7):595–596, 1963.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of Computation*, 48:203–209, 1987.

[LN83] R. Lidl and H. Niederreiter. *Finite Fields*, volume 20 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, Massachusetts, 1983.

[LV00a] A. Lenstra and E. Verheul. Selecting cryptographic key sizes. In *Public Key Cryptography — PKC 2000*. Springer-Verlag Lecture Notes in Computer Science, 2000.

[LV00b] A. Lenstra and E. Verheul. The XTR Public Key System. In *Lecture Notes in Computer Science 1880: Advances in Cryptology — CRYPTO 2000*, pages 1 – 19. Springer-Verlag, Berlin, 2000.

[Men93] A. J. Menezes. *Elliptic Curve Public Key Cryptosystems*. Kluwer Academic Publishers, 1993.

[Mih97] P. Mihăilescu. Optimal Galois field bases which are not normal. Fast Software Encryption rump session, 1997.

[Mil86] V. Miller. Uses of elliptic curves in cryptography. In *Lecture Notes in Computer Science 218: Advances in Cryptology — CRYPTO '85*, pages 417–426. Springer-Verlag, Berlin, 1986.

[Mon85] P. L. Montgomery. Modular Multiplication without Trial Division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[Mot93] Motorola Inc. *MC68000 8-/16-/32-bit Microprocessors User's Manual*. 1993.

[MvOV97] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1997.

[NIS00] NIST. Digital Signature Standard, FIPS Publication 186-2, 2000.

[NM96] D. Naccache and D. M'Raïhi. Cryptographic smart cards. *IEEE Micro*, 16(3):14–24, 1996.

[NMWdP95] D. Naccache, D. M'Raïhi, W. Wolfowicz, and A. di Porto. Are crypto-accelerators really inevitable? In *Advances in Cryptography — EUROCRYPT '95*, pages 404–409. Springer-Verlag Lecture Notes in Computer Science, 1995.

[SOOS95] R. Schroeppel, H. Orman, S. O'Malley, and O. Spatscheck. Fast key exchange with elliptic curve systems. *Advances in Cryptology — CRYPTO '95*, pages 43–56, 1995.

[WP01] A. Weimerskirch and C. Paar. An analysis of the karatsuba algorithm to multiply polynomials and its applications to parallel multipliers. Technical report, Cryptography and Information Security Group, ECE Department, WPI, 2001.

[YA95] S. Yeralan and A. Ahluwalia. *Programming and Interfacing the 8051 Microcontroller*. Addison-Wesley Publishing Company, 1995.